A Taste of Dynamic Information-Flow Control

15-136 Software Foundations of Security and Privacy

Arthur Azevedo de Amorim

October 17, 2019

Carnegie Mellon University

Motivation

Static information-flow control (IFC) is great

...but it has its shortcomings.

- Incompatible with dynamic languages (Python, JS, ...)
- Annotation burden can be a showstopper for legacy code
- Every bit of the program must handle security explicitly

Dynamic IFC to the rescue

Secrecy level of data is determined at *run time*, rather than statically. Thought to be impossible until 2009, when Sabelfeld and Russo proved noninterference for such a language.

Dynamic IFC to the rescue

Secrecy level of data is determined at *run time*, rather than statically. Thought to be impossible until 2009, when Sabelfeld and Russo proved noninterference for such a language.

Several advantages:

- More flexible and more permissive
- Easier for dynamic languages
- Simplifies migration of legacy code
- Security-aware code is more local

The Language

Syntax

Same as before, but with a *classification* expression.

$$
\begin{aligned}
e &:= x \mid n \mid e_1 + e_2 \mid e_1 \times e_2 \mid e@l \\
p, q &:= \text{true} \mid \text{false} \mid p \wedge q \mid p \vee q \mid \neg p \mid e_1 = e_2 \mid e_1 \leq e_2 \\
\alpha, \beta &:= x \leftarrow e \mid \alpha; \beta \mid \text{if } (p)\ \alpha \text{ else } \beta \mid \text{while } (p)\ \alpha
\end{aligned}
$$

Syntax

Same as before, but with a *classification* expression.

$$e := x \mid n \mid e_1 + e_2 \mid e_1 \times e_2 \mid e@l$$

$$p, q := \text{true} \mid \text{false} \mid p \wedge q \mid p \vee q \mid \neg p \mid e_1 = e_2 \mid e_1 \leq e_2$$

$$\alpha, \beta := x \leftarrow e \mid \alpha; \beta \mid \text{if } (p)\ \alpha \text{ else } \beta \mid \text{while } (p)\ \alpha$$

Meaning: the value of e at the secrecy level l.

Syntax

Same as before, but with a *classification* expression.

$$e := x \mid n \mid e_1 + e_2 \mid e_1 \times e_2 \mid e@l$$
$$p, q := \text{true} \mid \text{false} \mid p \wedge q \mid p \vee q \mid \neg p \mid e_1 = e_2 \mid e_1 \leq e_2$$
$$\alpha, \beta := x \leftarrow e \mid \alpha; \beta \mid \text{if } (p)\ \alpha \text{ else } \beta \mid \text{while } (p)\ \alpha$$

Meaning: the value of e at the secrecy level l.

(No typing rules are needed)

Semantics

Two main differences:

Semantics

Two main differences:

- Labels now exist during execution

Semantics

Two main differences:

- Labels now exist during execution
  - $\omega : \mathsf{Var} \to \mathbb{Z} \times L$
  - $\langle \omega, e \rangle \Downarrow_{\mathbb{Z}} n@l$, $\langle \omega, p \rangle \Downarrow_{\mathbb{B}} b@l$
  - $\langle \omega, l_{pc}, \alpha \rangle \Downarrow r$

Semantics

Two main differences:

- Labels now exist during execution
  - $\omega : \mathsf{Var} \rightarrow \mathbb{Z} \times L$
  - $\langle \omega, e \rangle \Downarrow_{\mathbb{Z}} n@l$, $\langle \omega, p \rangle \Downarrow_{\mathbb{B}} b@l$
  - $\langle \omega, l_{pc}, \alpha \rangle \Downarrow r$
- The result r is a final state or an error.

Semantics

Two main differences:

- Labels now exist during execution
    - $\omega : \mathsf{Var} \to \mathbb{Z} \times L$
    - $\langle \omega, e \rangle \Downarrow_{\mathbb{Z}} n@l$, $\langle \omega, p \rangle \Downarrow_{\mathbb{B}} b@l$
    - $\langle \omega, l_{pc}, \alpha \rangle \Downarrow r$
- The result r is a final state or an error.

Previous static checks generally become dynamic checks

Example: binary operators

Typing rule for static IFC:

$$\frac{\Gamma \vdash e_1 : l_1 \qquad \Gamma \vdash e_2 : l_2}{\Gamma \vdash e_1 \odot e_2 : l_1 \sqcup l_2}$$

Example: binary operators

Typing rule for static IFC:

$$\frac{\Gamma \vdash e_1 : l_1 \qquad \Gamma \vdash e_2 : l_2}{\Gamma \vdash e_1 \odot e_2 : l_1 \sqcup l_2}$$

Evaluation rule for dynamic IFC:

$$\frac{\langle \omega, e_1 \rangle \Downarrow_{\mathbb{Z}} n_1 @ l_1 \qquad \langle \omega, e_2 \rangle \Downarrow_{\mathbb{Z}} n_2 @ l_2}{\langle \omega, e_1 \odot e_2 \rangle \Downarrow_{\mathbb{Z}} (n_1 \odot n_2) @ (l_1 \sqcup l_2)}$$

Example: binary operators

Typing rule for static IFC:

$$\frac{\Gamma \vdash e_1 : l_1 \qquad \Gamma \vdash e_2 : l_2}{\Gamma \vdash e_1 \odot e_2 : l_1 \sqcup l_2}$$

Evaluation rule for dynamic IFC:

$$\frac{\langle \omega, e_1 \rangle \Downarrow_{\mathbb{Z}} n_1 @ l_1 \qquad \langle \omega, e_2 \rangle \Downarrow_{\mathbb{Z}} n_2 @ l_2}{\langle \omega, e_1 \odot e_2 \rangle \Downarrow_{\mathbb{Z}} (n_1 \odot n_2) @ (l_1 \sqcup l_2)}$$

Evaluating Programs

Expression evaluation

$$\frac{}{\langle \omega, x \rangle \Downarrow_{\mathbb{Z}} \omega(x)} \qquad \frac{}{\langle \omega, n \rangle \Downarrow_{\mathbb{Z}} n@\bot} \qquad \frac{\langle \omega, e \rangle \Downarrow_{\mathbb{Z}} n@l}{\langle \omega, e@l' \rangle \Downarrow_{\mathbb{Z}} n@(l \sqcup l')}$$

(Boolean evaluation is similar)

Assignments

Old typing rule

$$\frac{\Gamma \vdash e : l_e \qquad l_e \sqcup \Gamma(pc) \sqsubseteq \Gamma(x)}{\Gamma \vdash x \leftarrow e}$$

Assignments

Old typing rule

$$\frac{\Gamma \vdash e : l_e \qquad l_e \sqcup \Gamma(pc) \sqsubseteq \Gamma(x)}{\Gamma \vdash x \leftarrow e}$$

New eval rule, success

$$\frac{\langle \omega, e \rangle \Downarrow_{\mathbb{Z}} n@l_n \qquad \omega(x) = \_@l_x \qquad l_{pc} \sqsubseteq l_x}{\langle \omega, l_{pc}, x \leftarrow e \rangle \Downarrow \omega[x \mapsto n@(l_n \sqcup l_{pc})]}$$

New eval rule, error

$$\frac{\langle \omega, e \rangle \Downarrow_{\mathbb{Z}} n@l_n \qquad \omega(x) = \_@l_x \qquad l_{pc} \not\sqsubseteq l_x}{\langle \omega, l_{pc}, x \leftarrow e \rangle \Downarrow \text{error}}$$

Assignments

Old typing rule

$$\frac{\Gamma \vdash e : l_e \qquad l_e \sqcup \Gamma(pc) \sqsubseteq \Gamma(x)}{\Gamma \vdash x \leftarrow e}$$

New eval rule, success

$$\frac{\langle \omega, e \rangle \Downarrow_{\mathbb{Z}} n@l_n \qquad \omega(x) = \_@l_x \qquad l_{pc} \sqsubseteq l_x}{\langle \omega, l_{pc}, x \leftarrow e \rangle \Downarrow \omega[x \mapsto n@(l_n \sqcup l_{pc})]}$$

New eval rule, error

$$\frac{\langle \omega, e \rangle \Downarrow_{\mathbb{Z}} n@l_n \qquad \omega(x) = \_@l_x \qquad l_{pc} \not\sqsubseteq l_x}{\langle \omega, l_{pc}, x \leftarrow e \rangle \Downarrow \text{error}}$$

- Uses so-called *no-sensitive-upgrade check*; l_n is not used.

Assignments

Old typing rule

$$\frac{\Gamma \vdash e : l_e \qquad l_e \sqcup \Gamma(pc) \sqsubseteq \Gamma(x)}{\Gamma \vdash x \leftarrow e}$$

New eval rule, success

$$\frac{\langle \omega, e \rangle \Downarrow_{\mathbb{Z}} n@l_n \qquad \omega(x) = \_@l_x \qquad l_{pc} \sqsubseteq l_x}{\langle \omega, l_{pc}, x \leftarrow e \rangle \Downarrow \omega[x \mapsto n@(l_n \sqcup l_{pc})]}$$

New eval rule, error

$$\frac{\langle \omega, e \rangle \Downarrow_{\mathbb{Z}} n@l_n \qquad \omega(x) = \_@l_x \qquad l_{pc} \not\sqsubseteq l_x}{\langle \omega, l_{pc}, x \leftarrow e \rangle \Downarrow \text{error}}$$

- Uses so-called *no-sensitive-upgrade check*; l_n is not used.
- NB The label of x can go up!

Sequencing

$$\frac{\langle \omega_1, l_{pc}, \alpha \rangle \Downarrow \omega_2 \qquad \langle \omega_2, l_{pc}, \beta \rangle \Downarrow r}{\langle \omega_1, l_{pc}, \alpha; \beta \rangle \Downarrow r} \qquad \frac{\langle \omega_1, l_{pc}, \alpha \rangle \Downarrow \textsf{error}}{\langle \omega_1, l_{pc}, \alpha; \beta \rangle \Downarrow \textsf{error}}$$

Sequencing

$$\frac{\langle \omega_1, l_{pc}, \alpha \rangle \Downarrow \omega_2 \qquad \langle \omega_2, l_{pc}, \beta \rangle \Downarrow r}{\langle \omega_1, l_{pc}, \alpha; \beta \rangle \Downarrow r} \qquad \frac{\langle \omega_1, l_{pc}, \alpha \rangle \Downarrow \textsf{error}}{\langle \omega_1, l_{pc}, \alpha; \beta \rangle \Downarrow \textsf{error}}$$

Error stops execution

Conditionals

$$\frac{\langle \omega, p \rangle \Downarrow_{\mathbb{B}} \mathsf{true}\,@l_p \qquad \langle \omega, l_{pc} \sqcup l_p, \alpha \rangle \Downarrow r}{\langle \omega, l_{pc}, \mathsf{if}\ (p)\ \alpha\ \mathsf{else}\ \beta \rangle \Downarrow r}$$

$$\frac{\langle \omega, p \rangle \Downarrow_{\mathbb{B}} \mathsf{false}\,@l_p \qquad \langle \omega, l_{pc} \sqcup l_p, \alpha \rangle \Downarrow r}{\langle \omega, l_{pc}, \mathsf{if}\ (p)\ \alpha\ \mathsf{else}\ \beta \rangle \Downarrow r}$$

Looping

$$\frac{\langle \omega, p \rangle \Downarrow_{\mathbb{B}} \textsf{true}\,@l_p \qquad \langle \omega, l_{pc} \sqcup l_p, \alpha; \textsf{while}\ (p)\ \alpha \rangle \Downarrow r}{\langle \omega, l_{pc}, \textsf{while}\ (p)\ \alpha \rangle \Downarrow r}$$

$$\frac{\langle \omega, p \rangle \Downarrow_{\mathbb{B}} \textsf{false}\,@l_p}{\langle \omega, l_{pc}, \textsf{while}\ (p)\ \alpha \rangle \Downarrow \omega}$$

Example program

```
y <- true@L;
z <- true@L;
if (x) {y <- false@L};
if (y) {z <- false@L};
```

Example program

```
y <- true@L;
z <- true@L;
if (x) {y <- false@L};
if (y) {z <- false@L};
```

- What happens when `x = true@H` and when `x = false@H`?

Example program

```
y <- true@L;
z <- true@L;
if (x) {y <- false@L};
if (y) {z <- false@L};
```

- What happens when `x = true@H` and when `x = false@H`?
- Can we write this program in the static language?

Noninterference

Statement

If a program α

- runs on equivalent states $\omega_1 \approx_l \omega_2$, and
- both runs *successfully* terminate: $\langle \omega_1, l_{pc}, \alpha \rangle \Downarrow \omega_1'$ and $\langle \omega_2, l_{pc}, \alpha \rangle \Downarrow \omega_2'$

then the final states are equivalent $\omega_1' \approx_l \omega_2'$.

What Does Equivalence Mean?

$$\frac{l_n \sqsubseteq l}{n@l_n \approx_l n@l_n} \qquad \frac{l_1 \not\sqsubseteq l \qquad l_2 \not\sqsubseteq l}{n_1@l_1 \approx_l n_2@l_2} \qquad \frac{\forall x.\ \omega_1(x) \approx_l \omega_2(x)}{\omega_1 \approx_l \omega_2}$$

Lemmas

- Expression evaluation respects equivalence: if $\omega_1 \approx_l \omega_2$ and $\langle \omega_i, e \rangle \Downarrow_{\mathbb{Z},\mathbb{B}} r_i$, then $r_1 \approx_l r_2$.
- Confinement: if $\langle \omega, l_{pc}, \alpha \rangle \Downarrow \omega'$ and $l_{pc} \not\sqsubseteq l$, then $\omega' \approx_l \omega$.
- $\approx_l$ is an equivalence relation.

Use induction on the execution length.

Wrapping up

IFC in the Wild

- Static IFC languages: Jif (based on Java), FlowCaml (based on OCaml), SPARK (based on Ada), ...
- Research IFC OSs: HiStar, Asbestos, ...
- Taint tracking in Perl, Ruby, etc
- Dynamic IFC languages and libraries: JavaScript monitors, LIO (Haskell)

- Static IFC languages: Jif (based on Java), FlowCaml (based on OCaml), SPARK (based on Ada), ...
- Research IFC OSs: HiStar, Asbestos, ...
- Taint tracking in Perl, Ruby, etc
- Dynamic IFC languages and libraries: JavaScript monitors, LIO (Haskell)

Demo!

Thank you for your attention!

Some references:

- "From Dynamic to Static and Back: Riding the Roller Coaster of Information-Flow Control Research." Sabelfeld and Russo, 2009. (First dynamic IFC language with a proof of noninterference.)
- "Flexible Dynamic Information Flow Control in Haskell." Stefan et al., 2011.